# Problem 1:

**Print Matrix In Spiral Order**

Given a character matrix, return all the characters in the clockwise spiral order starting from the top-left.

**Example**

Input:

[

['X' 'Y' 'A']

['M' 'B' 'C']

['P' 'Q' 'R']

]

Output: "XYACRQPMB"

For the given matrix rows = 3 and cols = 3. Spiral order is 'X' -> 'Y' -> 'A' -> 'C' -> 'R' -> 'Q' -> 'P' -> 'M' -> 'B'. So return string "XYACRQPMB" of length rows \* cols = 9.

Output: Return a string *res*, of length *rows* \* *cols* denoting the spiral order of *matrix*.

Constraints:

* 1 <= *rows*, *cols*
* 1 <= *rows* \* *cols* <= 10^5
* Any character in *matrix* will be either uppercase letter ('A' - 'Z') or lowercase letter ('a' - 'z').
* Avoid recursion.

This problem is less about logic, but more about careful index manipulation.

Hint - It may be faster to write this, if you name your variables clearly. Instead of i,j,k,l etc, try naming them like row, col, start, end etc. That will also help your interviewer follow along more easily.

# Solution

There are many solutions possible for this problem.

Here we will provide one interesting solution that uses only one for loop. Imagine there is a turtle standing at point (0,0), that is, top-left corner, facing east (to the right). It will keep going forward and each time it sees a sign, the turtle will turn right. So if we put the turtle at point (0,0) facing right-ward, and if we place the signs at appropriate places, the turtle will traverse the array in a spiral way. Now the problem is: "Where do we put the signs?"

Let's see where we should put the signs (marked by #, and numbers by O):

For a grid that looks like this:

O O O O

O O O O

O O O O

O O O O

We put the signs like this:

O O O #

# O # O

O # # O

# O O #

For a grid that looks like this:

O O O

O O O

O O O

O O O

We put the signs like this:

O O #

# # O

O # O

# O #

And for a grid that looks like this:

O O O O O O O

O O O O O O O

O O O O O O O

O O O O O O O

O O O O O O O

We put the signs like this:

O O O O O O #

# O O O O # O

O # O O # O O

O # O O O # O

# O O O O O #

We can divide the grid in 4 parts and then follow some patterns.

4 parts will be:

1) top-left (lets call it a)

2) top-right (lets call it b)

3) bottom-right (lets call it c)

4) bottom-left (lets call it d)

So 6 \* 6 grid will be divided like:

a a a b b b

a a a b b b

a a a b b b

d d d c c c

d d d c c c

d d d c c c

Now for most of the points we can easily decide in which part they will fall, except points which are horizontally centered or vertically centered. Horizontally centered points: Consider them in top parts. Vertically centered points: Consider them in right parts.

So 5 \* 7 grid will be divided like:

a a a b b b b

a a a b b b b

a a a b b b b

d d d c c c c

d d d c c c c

Now again look at the grid:

O O O O O O #

# O O O O # O

O # O O # O O

O # O O O # O

# O O O O O #

and try to find patterns from parts:

O O O O O O #

# O O O O # O

O # O O # O O

O # O O O # O

# O O O O O #

For top-right, bottom-right and bottom-left pattern is same!

If matrix size is rows \* cols then for any point (at position cur\_row and cur\_col) if we want to check if there is a sign or not simply check:

1) top-right: cur\_row == cols - 1 - cur\_col

2) bottom-right: rows - 1 - cur\_row == cols - 1 - cur\_col

3) bottom-left: rows - 1 - cur\_row == cur\_col

We can write conditions separately or combine them as:

min(cur\_row, rows - 1 - cur\_row) == min(cur\_col, cols - 1 - cur\_col) ......(1)

Now for the top-left part we need to check:

cur\_row == cur\_col + 1 ......(2)

Now you know where to put the signs! How to check if point is in top-left or other parts?

/\*

Consider these grids to understand what the below code does.

O O O O O O #

# O O O O # O

O # O O # O O

O # O O O # O

# O O O O O #

=

O O O O O O #

# O O O O # O

O # O O # O O

O # O O O # O

# O O O O O #

< (rows + 1) / 2 will give priority to top part when current position is horizontally centered.

< cols / 2 will give priority to right part when current position is vertically centered.

\*/

if ((cur\_row < (rows + 1) / 2) && (cur\_col < cols / 2))

{

// Condition to turn when current position is in top-left part.

}else{

// Condition to turn when current position in other parts.

}

**Time Complexity:**

O(rows \* cols).

We are traversing the whole vector once.

**Auxiliary Space Used:**

O(1).

**Space Complexity:**

O(rows \* cols).

# Problem 2: Reverse the string without reversing words

**Reverse The Ordering Of Words In A String**

Given a string *s* containing a set of words, transform it such that the words appear in the reverse order. Words in *s* are separated by one or more spaces.

**Example One**

Input: “I will do it.”

Output: “it. do will I”

**Example Two**

Input: "   word1 word2 " (Note: there are 3 spaces in the beginning, 2 spaces between the words and 1 space at the end.)

Output: " word2  word1 " (Note: there is 1 space in the beginning, 2 spaces between the words and 3 spaces at the end.)

**Example Three**

Input: "word1, word2;"

Output: "word2; word1,"

Constraints:

* 1 <= length of *s* <= 10^5
* *s* contains only lowercase and uppercase alphabetical characters, spaces and punctuation marks ".,?!':;-" (quotes not included).

Punctuation marks are considered a part of the word.

Usage of built-in string functions is NOT allowed.

An in-place linear solution is expected.

For languages that have immutable strings, convert the input string into a character array and work in-place on that array. Convert it back to the string before returning. Ignore the extra linear space used in that conversion, as long as you're only using constant space after conversion to character array.

Trivia: This is a very old interview question. Google used it as one of their qualifier questions in Google CodeJam in the past, too.

# Solution

One idea for the solution is:

1) Reverse the whole string.

2) Then reverse the individual words.

For example, if the input is:

s = "Have a nice day!"

1) Then first reverse the whole string,

s = "!yad ecin a evaH"

2) Then reverse the individual words,

s = "day! nice a Have"

**Time Complexity:**

O(n).

The first pass over the string is obviously O(n/2) = O(n). The second pass is O(n + combined length of all words / 2) = O(n + n/2) = O(n), which makes this an O(n) algorithm.

**Auxiliary Space used:**

O(1).

**Space Complexity:**

O(n).

# Problem 3: